コンパイラに関する実験

作成日：2006/01/31
作成者：松本 亮介
学部：工学部
学科；情報工学科
学籍番号：1030160322

§§§コンパイラに関する実験§§§

作成日　2006/1/31

情報工学科　1031060322　松本亮介

１、プログラムと各部の説明

作成したコンパイラのプログラムの説明を行う。

最初に作成したコンパイラのプログラムを記述しておく。

プログラムソース'compiler.c'ここから↓

```
/*-------------------------------------------------------------------------------------------------
                                四則演算を解析するコンパイラ
-------------------------------------------------------------------------------------------------
                                作成者:松本亮介(Ryousuke Matsumoto)
                                作成日:2006.01.31
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
                                ファイル名: compiler.c
-------------------------------------------------------------------------------------------------*/

#include <stdio.h>
#include<stdlib.h>
#include <ctype.h>

#define MAXLENG 128
#define MAXLENGS 256

/* ------------サブルーチンここから--------------- */
int lexical_analysis(void);/* 字句解析関数 */
int syntactic_analysis_1(void);/* 構文解析関数１ */
void syntactic_analysis_2(void);/* 構文解析関数２ */
void syntactic_analysis_3(void);/* 構文解析関数３ */
void syntactic_analysis_4(void);/* 構文解析関数４ */
int code_generation(void);/* コード生成関数 */
/* ------------------------------------------- */
/* ---------グローバル宣言--------- */
```

```
FILE *fp,*fp2;

/* ----------字句解析の字句と種類を格納する構造体ここから-------- */
struct character{
        char jiku[MAXLENG];
        int result;
}lexical_result[MAXLENG],syntactic_result[MAXLENG];
/* ---------字句解析の字句と種類を格納する構造体ここまで--------- */

        int i=0,j=0,n,o;
        int num,gyou_kaihi;
        int stack[MAXLENGS];
        char op,*temp[MAXLENGS];
/* ---------------------------------- */

/*   ------------------------------------  メ イ ン ル ー チ ン こ こ か ら
------------------------------------------ */
int main(int argc,char *argv[]){
        int ni;

        if((fp=fopen(argv[1],"r"))==NULL){
                printf("Cannot Open File.¥n");
                exit(1);
        }
        fp2=fopen("compiler_data.txt","w");
        ni=1;
        do{
                while(op!=';'){
                        gyou_kaihi=ni;
                        fprintf(fp2,"【%d-gyoume】¥n",ni);
                        ni++;
                        fprintf(fp2,"【motono-siki】¥n");
                        lexical_analysis();/* 字句解析関数呼び出し */
                        fprintf(fp2,"¥n");
                        syntactic_analysis_1();/* 構文解析関数呼び出し */
                        code_generation();/* コード生成関数呼び出し */
                }
                op=getc(fp);
        }while(op!=EOF);
```

```
        printf("¥nIt succeeded in making 'compiler_data.txt'.¥n");
        return 0;
}
/*  ---------------------------------------- メインルーチンここまで
---------------------------------------- */

/*  ---------------------------------- サブルーチン（字句解析）ここから
---------------------------------- */
/* ----------------------------------
・構造体の result に字句の種類を格納
・構造体の jiku に字句を格納
---------------------------------- */
int lexical_analysis(){
        int d,h,li,li2;
        d=0,h=0;

        for(li=0;li<MAXLENG;li++){
                lexical_result[li].result=0;
                for(li2=0;li2<MAXLENG;li2++){
                        lexical_result[li].jiku[li2]=' ';
                }
        }
        n=1;
        o=1;
        op=getc(fp);
        while(1){
                if(isspace(op)){/* スペースの判定 */
                        putc(op,fp2);
                        op=getc(fp);
                }
                else if(isalpha(op)){/* 識別子の判定 */
                        lexical_result[d].result=1;
                        lexical_result[d].jiku[h]=op;
                        op=getc(fp);
                        h++;
                        while(isalnum(op)){
                                lexical_result[d].jiku[h]=op;
                                op=getc(fp);
                                h++;
                        }
```

```
                fprintf(fp2,"i%d.%d",n,gyou_kaihi);
                n++;
                d++;
                h=0;
        }
        else if(isdigit(op)){/* 定数の判定 */
                lexical_result[d].result=2;
                lexical_result[d].jiku[h]=op;
                op=getc(fp);
                h++;
                while(isdigit(op)){
                        lexical_result[d].jiku[h]=op;
                        op=getc(fp);
                        h++;
                }
                fprintf(fp2,"num%d.%d",o,gyou_kaihi);
                o++;
                d++;
                h=0;
        }
        else if(op==':'){/* ':='の判定 */
                lexical_result[d].result=3;
                lexical_result[d].jiku[h]=op;
                putc(op,fp2);
                op=getc(fp);
                h++;
                if(op=='='){
                        lexical_result[d].jiku[h]=op;
                        putc(op,fp2);
                        op=getc(fp);
                        d++;
                        h=0;
                }
                else{
                        printf("lexical ERROR!¥n");/* ':'の次に'='がこなかったときエラー
表示 */
                }
        }
        else if(op=='+'){/* '+'の判定 */
                lexical_result[d].result=4;
```

```
        lexical_result[d].jiku[h]=op;
        putc(op,fp2);
        op=getc(fp);
        d++;
        h=0;
}
else if(op=='-'){/* '-'の判定 */
        lexical_result[d].result=5;
        lexical_result[d].jiku[h]=op;
        putc(op,fp2);
        op=getc(fp);
        d++;
        h=0;
}
else if(op=='*'){/* '*'の判定 */
        lexical_result[d].result=6;
        lexical_result[d].jiku[h]=op;
        putc(op,fp2);
        op=getc(fp);
        d++;
        h=0;
}
else if(op=='/'){/* '/'の判定 */
        lexical_result[d].result=7;
        lexical_result[d].jiku[h]=op;
        putc(op,fp2);
        op=getc(fp);
        d++;
        h=0;
}
else if(op=='('){/* '('の判定 */
        lexical_result[d].result=8;
        lexical_result[d].jiku[h]=op;
        putc(op,fp2);
        op=getc(fp);
        d++;
        h=0;
}
else if(op==')'){/* ')'の判定 */
        lexical_result[d].result=9;
```

```
                        lexical_result[d].jiku[h]=op;
                        putc(op,fp2);
                        op=getc(fp);
                        d++;
                        h=0;
                }
                else if(op==';'){/* ';'の判定 */
                        lexical_result[d].result=10;
                        lexical_result[d].jiku[h]=op;
                        break;
                }
        }
        return 0;
}
/*  ------------------------------------ サブルーチン（字句解析）ここまで
---------------------------------- */


/*  ------------------------------------ サブルーチン（構文解析１）ここから
---------------------------------- */
int syntactic_analysis_1(){
        int m,l,gyou1;
        char *x;

        for(m=0;m<MAXLENG;m++){
                for(l=0;l<MAXLENG;l++){
                        syntactic_result[m].jiku[l]=' ';
                }
        }



        i=0;
        j=0;
        l=0;
        n=1;
        o=1;
        fprintf(fp2,"¥n【gyakupooranndokihou-siki】¥n");
        while(lexical_result[i].result!=10){
/* １文字目の字句の構文解析の判定 */
                if(lexical_result[i].result==1){
```

```
                        syntactic_result[j].result=1;
                        for(m=0;m<MAXLENG;m++){
                                if(isalnum(lexical_result[i].jiku[m])){
                                        syntactic_result[j].jiku[m]=lexical_result[i].jiku[m];
                                }
                        }
                        fprintf(fp2,"i%d.%d ",n,gyou_kaihi);
                        n++;
                        j++;
                }
                i++;

/* ２文字目以降の字句の構文解析の判定 */

        if(lexical_result[i].result==2||lexical_result[i].result==3||lexical_result[i].result==4
||lexical_result[i].result==5||

        lexical_result[i].result==6||lexical_result[i].result==7||lexical_result[i].result==8||l
exical_result[i].result==9){
                        syntactic_analysis_4();
                        syntactic_analysis_3();
                        syntactic_analysis_2();
                }

                if(lexical_result[i].result==10){
                        syntactic_result[j].result=3;
                        fprintf(fp2,":= ");
                        break;
                }
        }

        fprintf(fp2,"¥n¥n【kigouhyou/teisuuhyou】¥n");/* 記号表・定数表の作成 */
        fprintf(fp2,"kigou1¥t¥tkigou2¥t¥taddress¥n");/* 記号表の作成 */

        n=1;
        for(m=0;m<MAXLENG;m++){
                if(syntactic_result[m].result==1){
                        while(isalnum(syntactic_result[m].jiku[l])){
                                putc(syntactic_result[m].jiku[l],fp2);
                                l++;
```

```
                        }
                        fprintf(fp2,"¥t¥ti%d.%d",n,gyou_kaihi);
                        n++;
                        x=syntactic_result[m].jiku;
                        if(gyou_kaihi!=0){/* アドレスの重複を防ぐために、行数によってアドレスをす
すめておく */
                                for(gyou1=0;gyou1<=gyou_kaihi;gyou1++){
                                        x++;
                                }
                        }
                        temp[m]=x;
                        fprintf(fp2,"¥t¥t%p¥n",x);
                        l=0;
                }
        }
        fprintf(fp2,"¥nteisu1¥t¥tteisu2¥t¥taddress¥n");/* 定数表の作成 */
                n=1;
                for(m=0;m<MAXLENG;m++){
                        if(syntactic_result[m].result==2){
                                while(isdigit(syntactic_result[m].jiku[l])){
                                        putc(syntactic_result[m].jiku[l],fp2);
                                        l++;
                                }
                                fprintf(fp2,"¥t¥tnum%d.%d",n,gyou_kaihi);
                                n++;
                                x=syntactic_result[m].jiku;
                                if(gyou_kaihi!=0){/* アドレスの重複を防ぐために、行数によってアド
レスをすすめておく */
                                        for(gyou1=0;gyou1<=gyou_kaihi*3;gyou1++){
                                                x++;
                                        }
                                }
                                temp[m]=x;
                                fprintf(fp2,"¥t¥t%p¥n",x);
                                l=0;
                        }
                }
                fprintf(fp2,"¥n");
                return 0;
}
```

```
/*  ---------------------------------- サブルーチン（構文解析１）ここまで
---------------------------------- */

/*

E → TE'
E' → +T[+]E'|ε
E' → -T[-]E'|ε
T → FT'
T' → *F[*]T'|ε
T' → /F[/]T'|ε
F → (E)|i[i]|num[num]

*/

/*
E'がsyntactic_analysis_2()関数に対応
T'がsyntactic_analysis_3()関数に対応
F がsyntactic_analysis_4()関数に対応
*/

/*  ---------------------------------- サブルーチン（構文解析２）ここから
---------------------------------- */
void syntactic_analysis_2()
{
        if(lexical_result[i].result==4){/* '+'の構文解析の判定 */
                i++;
                syntactic_analysis_4();
                syntactic_analysis_3();
                syntactic_result[j].result=4;
                fprintf(fp2,"+ ");
                j++;
                syntactic_analysis_2();
        }
        else if(lexical_result[i].result==5){/* '-'の構文解析の判定 */
                i++;
                syntactic_analysis_4();
                syntactic_analysis_3();
                syntactic_result[j].result=5;
                fprintf(fp2,"- ");
```

```
                j++;
                syntactic_analysis_2();
        }
}
/*  ---------------------------------- サブルーチン（構文解析２）ここまで
---------------------------------- */


/*  ---------------------------------- サブルーチン（構文解析３）ここから
---------------------------------- */
void syntactic_analysis_3()
{
        if(lexical_result[i].result==6){/* '*'の構文解析の判定 */
                i++;
                syntactic_analysis_4();
                syntactic_result[j].result=6;
                fprintf(fp2,"* ");
                j++;
                syntactic_analysis_3();
        }
        else if(lexical_result[i].result==7){/* '/'の構文解析の判定 */
                i++;
                syntactic_analysis_4();
                syntactic_result[j].result=7;
                fprintf(fp2,"/ ");
                j++;
                syntactic_analysis_3();
        }
}
/*  ---------------------------------- サブルーチン（構文解析３）ここまで
---------------------------------- */


/*  ---------------------------------- サブルーチン（構文解析４）ここから
---------------------------------- */
void syntactic_analysis_4(){
        int t;

        if(lexical_result[i].result==1){/* '識別子'の構文解析の判定 */
                syntactic_result[j].result=1;
```

```
		for(t=0;t<MAXLENG;t++){
			if(isalnum(lexical_result[i].jiku[t])){
				syntactic_result[j].jiku[t]=lexical_result[i].jiku[t];
			}
		}
		fprintf(fp2,"i%d.%d ",n,gyou_kaihi);
		n++;
		j++;
		i++;
	}
	else if(lexical_result[i].result==2){/* '定数'の構文解析の判定 */
		syntactic_result[j].result=2;
		for(t=0;t<MAXLENG;t++){
			if(isdigit(lexical_result[i].jiku[t])){
				syntactic_result[j].jiku[t]=lexical_result[i].jiku[t];
			}
		}
		fprintf(fp2,"num%d.%d ",o,gyou_kaihi);
		o++;
		j++;
		i++;
	}
	else if(lexical_result[i].result==8){/* '('と')'の構文解析の判定 */
		syntactic_result[j].result=8;
		i++;
		j++;
		syntactic_analysis_4();
		syntactic_analysis_3();
		syntactic_analysis_2();
		if(lexical_result[i].result==9){
			syntactic_result[j].result=9;
			i++;
			j++;
		}
	}
}
/* ---------------------------------- サブルーチン（構文解析４）ここまで
---------------------------------- */
```

```
/*  ---------------------------------- サブルーチン（コード生成）ここから
---------------------------------- */
int code_generation(){
/*
・構造体の kind に ACC の時は'-1'そうでないときは'1'を格納。
・構造体の data にアドレスを格納。
 */
        struct stacks{
                int kind;
                char data;
        }stack[MAXLENG];

        int p,q,r,gyou2;
        char *str;

        q=0;
        fprintf(fp2,"【code-generation】¥n");
        if(syntactic_result[0].result==1||syntactic_result[0].result==2){/* 一つめのスタックに格
納 */
                str=syntactic_result[0].jiku;
                stack[0].kind=1;
                stack[0].data=(int)&str;
        }
        else{
                printf("code_generation ERROR!¥n");/* 一つめのスタックに識別子か定数が格納されな
かった場合エラー表示 */
        }

        for(p=1;p<MAXLENG;p++){
                if(syntactic_result[p].result==1||syntactic_result[p].result==2){/* 識別子か定数
の場合はスタックにアドレスをプッシュダウンしていく */
                        if(stack[q].kind==0){
                                str=syntactic_result[p].jiku;
                                stack[q].kind=1;
                                stack[q].data=(int)&str;
                                break;
                        }
                        q++;
                        str=syntactic_result[p].jiku;
                        stack[q].kind=1;
```

```
                        stack[q].data=(int)&str;
                        if(gyou_kaihi!=0){
                                for(gyou2=0;gyou2<=gyou_kaihi*MAXLENG;gyou2++){
                                        str++;
                                }
                        }
                }
/* ------------------------------------------------------------
スタック[i]とスタック[i-1]が演算数が記憶されたアドレスである時
------------------------------------------------------------ */
                else if(stack[q].kind==1 && stack[q-1].kind==1){
                        if(syntactic_result[p].result==3){/* ':='のコード生成の判定 */
                                for(r=0;r<q;r++){/* スタック内にACCの存在を検索 */
                                        if(stack[r].kind==-1){
                                                stack[r].kind=1;
                                                stack[r].data=(int)&str;
                                                fprintf(fp2,"STORE¥t¥t%p¥n",str);
                                                str++;
                                                break;
                                        }
                                }
                                fprintf(fp2,"LOAD¥t¥t%p¥n",temp[p-1]);
                                fprintf(fp2,"STORE¥t¥t%p¥n",temp[p-2]);
                                stack[q].kind=0;
                                stack[--q].kind=-1;
                        }

                        if(syntactic_result[p].result==4){/* '+'のコード生成の判定 */
                                for(r=0;r<q;r++){
                                        if(stack[r].kind==-1){
                                                stack[r].kind=1;
                                                stack[r].data=(int)&str;
                                                fprintf(fp2,"STORE¥t¥t%p¥n",str);
                                                str++;
                                                break;
                                        }
                                }
                                fprintf(fp2,"LOAD¥t¥t%p¥n",temp[p-2]);
                                fprintf(fp2,"ADD¥t¥t%p¥n",temp[p-1]);
                                stack[q].kind=0;
```

```
        stack[--q].kind=-1;
    }
    if(syntactic_result[p].result==5){/* '-'のコード生成の判定 */
        for(r=0;r<q;r++){
            if(stack[r].kind==-1){
                stack[r].kind=1;
                stack[r].data=(int)&str;
                fprintf(fp2,"STORE¥t¥t%p¥n",str);
                str++;
                break;
            }
        }
        fprintf(fp2,"LOAD¥t¥t%p¥n",temp[p-2]);
        fprintf(fp2,"SUB¥t¥t%p¥n",temp[p-1]);
        stack[q].kind=0;
        stack[--q].kind=-1;
    }
    if(syntactic_result[p].result==6){/* '*'のコード生成の判定 */
        for(r=0;r<q;r++){
            if(stack[r].kind==-1){
                stack[r].kind=1;
                stack[r].data=(int)&str;
                fprintf(fp2,"STORE¥t¥t%p¥n",str);
                str++;
                break;
            }
        }
        fprintf(fp2,"LOAD¥t¥t%p¥n",temp[p-2]);
        fprintf(fp2,"MUL¥t¥t%p¥n",temp[p-1]);
        stack[q].kind=0;
        stack[--q].kind=-1;
    }
    if(syntactic_result[p].result==7){/* '/'のコード生成の判定 */
        for(r=0;r<q;r++){
            if(stack[r].kind==-1){
                stack[r].kind=1;
                stack[r].data=(int)&str;
                fprintf(fp2,"STORE¥t¥t%p¥n",str);
                str++;
                break;
```

```
                                        }
                                }
                                fprintf(fp2,"LOAD¥t¥t%p¥n",temp[p-2]);
                                fprintf(fp2,"DIV¥t¥t%p¥n",temp[p-1]);
                                stack[q].kind=0;
                                stack[--q].kind=-1;
                        }
                }
/* ----------------------------------------------------------------------------------------
スタック[i]が演算数が記憶されたアドレスで、スタック[i-1]が演算数がACCに記憶さてれている時
---------------------------------------------------------------------------------------- */
                else if(stack[q].kind==1 && stack[q-1].kind==-1){
                        if(syntactic_result[p].result==3){
                                fprintf(fp2,"STORE¥t¥t%p¥n",temp[p-1]);/* ':='のコード生成の判定
*/
                                stack[q].kind=0;
                                stack[--q].kind=-1;
                        }
                        if(syntactic_result[p].result==4){/* '+'のコード生成の判定 */
                                fprintf(fp2,"ADD¥t¥t%p¥n",temp[p-1]);
                                stack[q].kind=0;
                                stack[--q].kind=-1;
                        }
                        if(syntactic_result[p].result==5){/* '-'のコード生成の判定 */
                                fprintf(fp2,"SUB¥t¥t%p¥n",temp[p-1]);
                                stack[q].kind=0;
                                stack[--q].kind=-1;
                        }
                        if(syntactic_result[p].result==6){/* '*'のコード生成の判定 */
                                fprintf(fp2,"MUL¥t¥t%p¥n",temp[p-1]);
                                stack[q].kind=0;
                                stack[--q].kind=-1;
                        }
                        if(syntactic_result[p].result==7){/* '/'のコード生成の判定 */
                                fprintf(fp2,"DIV¥t¥t%p¥n",temp[p-1]);
                                stack[q].kind=0;
                                stack[--q].kind=-1;
                        }
                }
/* ----------------------------------------------------------------------------------------
```

```
スタック[i]が演算数がACCに記憶さてれていて、スタック[i-1]が演算数が記憶されたアドレスの時
---------------------------------------------------------------------------------------- */
        else if(stack[q].kind==-1 && stack[q-1].kind==1){
                if(syntactic_result[p].result==3){/* ':='のコード生成の判定 */
                        fprintf(fp2,"STORE¥t¥t%p¥n",temp[q-1]);
                        stack[q].kind=0;
                        stack[--q].kind=-1;
                }
                if(syntactic_result[p].result==4){/* '+'のコード生成の判定 */
                        str--;
                        fprintf(fp2,"ADD¥t¥t%p¥n",str);
                        str++;
                        stack[q].kind=0;
                        stack[--q].kind=-1;
                }
                if(syntactic_result[p].result==5){/* '-'のコード生成の判定 */
                        fprintf(fp2,"STORE¥t¥t%p¥n",str);
                        str--;
                        fprintf(fp2,"LOAD¥t¥t%p¥n",str);
                        str++;
                        fprintf(fp2,"SUB¥t¥t%p¥n",str);
                        str++;
                        stack[q].kind=0;
                        stack[--q].kind=-1;
                }
                if(syntactic_result[p].result==6){/* '*'のコード生成の判定 */
                        str--;
                        fprintf(fp2,"MUL¥t¥t%p¥n",str);
                        str++;
                        stack[q].kind=0;
                        stack[--q].kind=-1;
                }
                if(syntactic_result[p].result==7){/* '/'のコード生成の判定 */
                        fprintf(fp2,"STORE¥t¥t%p¥n",str);
                        str--;
                        fprintf(fp2,"LOAD¥t¥t%p¥n",str);
                        str++;
                        fprintf(fp2,"DIV¥t¥t%p¥n",str);
                        str++;
                        stack[q].kind=0;
```

```
                                stack[--q].kind=-1;
                        }
                }
        }
        fprintf(fp2,"¥n¥n¥n");
        return 0;
}
/*  ---------------------------------- サブルーチン（コード生成）ここまで
---------------------------------- */
```

プログラムソース'compiler.c'ここまで↑

まずは、サブルーチンから。
lexical_analysis・・・字句解析関数
syntactic_analysis1~4・・・構文解析関数
code_generarion・・・コード生成関数

まずは、全体を通して使う構造体として、

lexical_result[].jiku
lexical_result[].result
syntactic_result[].jiku
syntactic_result[].result

を宣言しておく。

順次、lexical_analysis 関数の説明からしていく。

（１）、lexical_analysis()関数
.jiku には、解析した字句を格納する。
.result には、解析した字句の種類を格納する。

ここでは、字句の種類を以下の数字に置き換えて格納していく。

識別子のときは、'1'
定数のときは、'2'

':='のときは、'3'
'+'のときは、'4'

'-'のときは、'5'
'*'のときは、'6'
'/'のときは、'7'
'('のときは、'8'
')'のときは、'9'
';'のときは、'10'

対象言語のテキストファイルより、一文字づつ取り出し、'op'に格納して、その文字の判定を行う。

（２)、syntactic_analysis_1()関数
lexical_result[].result に格納されている結果にしたがって、順次構文解析を行っていく。

配列[i]があるとして、
lexical_result[i].result の格納されている値を、配列の数[i]に対応して、syntactic_result[i].result にそれぞれ格納していく。

同様に、
lexical_result[i].jiku の格納されている値を、配列の数[i]に対応して、syntactic_result[i].jiku にそれぞれ格納していく。

syntactic_analysis_2()関数、syntactic_analysis_3()関数、syntactic_analysis_4()関数、

を通過し、構文解析によって振り分けられた字句に識別子や定数に対応するアドレスを、記号表と定数表に表示する。

その際に、以下のプログラムソースで、対象言語の１行目から、次の行の対象言語を読み込んだときに、アドレスが重複してしまわないように、１行目のアドレスの振り分けが終わった時点で、先頭で使用したアドレスから行数に対応してアドレスを進めておくことで、重複を避けている。

```
x=syntactic_result[m].jiku;
if(w!=0){
        for(gyou1=0;gyou1<=w;gyou1++){
                x++;
        }
}
temp[m]=x;
```

また、この関数の時点で、コード生成で用いるアドレスを temp[]に格納している。
syntactic_result[i].result と lexical_result[i].result と temp[i]の値は、それぞれのものと一貫性があ

るように格納していく。

（３）、syntactic_analysis_2()関数,syntactic_analysis_3()関数,syntactic_analysis_4()関数
これらの関数の基本的な仕組みは、以下の生成規則に対応している。

E → TE'
E' → +T[+}E'|ε
E' → -T[-}E'|ε
T → FT'
T' → *F[*]T'|ε
T' → /F[/]T'|ε
F → (E)|i[i]|num[num]

この生成規則の、

E'が syntactic_analysis_2()関数に対応
T'が syntactic_analysis_3()関数に対応
F が syntactic_analysis_4()関数に対応

している。

この生成規則にしたがって、syntactic_analysis_1()関数～syntactic_analysis_4()関数の中を字句に対応して関数内を通過し、構文解析を行っていく。

（４）、code_generation()関数
コード生成を行う関数である。

stack[].kind・・・ACC の時は'-1'を、そうでないときは'1'を格納する。

そして、この stack[].kind に従ってコード生成を行う。
sytactic[].result に格納されている値（演算記号の種類）と、コード生成におけるコードは以下のように対応して生成される。

'3'・・・STORE
'4'・・・ADD
'5'・・・SUB
'6'・・・MUL
'7'・・・DIV

さらに、stack[].kind に格納されている値（ACC の有無）に従って、以下のようなコード生成の場合分けがされている。

（ｉ）stack[i]==1、stack[i-1]==1 の時、
stack[i]のアドレスが V1、stack[i-1]のアドレスが V2 であれば、その時の sytactic[].result に格納されている値に従って、以下のようなコードが生成される。

```
STORE   T       （ただし、stack[1,....,i-2]の中に'-1'がある場合）
LOAE    V2
コード  V1      （コードは sytactic[].result の値に従う）
```

また、x := 3;（x 3 :=）のような対象言語の場合は、x のアドレスが V2、3 のアドレスが V1 となるので、以下のようなコードが生成される。

```
LOAD    V1
STORE   V2
```

（ⅱ）stack[i]==1、stack[i-1]==-1 の時、
stack[i]のアドレスが V1 であれば、sytactic[].result に格納されている値に従って、以下のようなコードが生成される。

```
コード  V1      （コードは sytactic[].result の値に従う）
```

（ⅲ）stack[i]==-1、stack[i-1]==1 の時、
stack[i-1]のアドレスが V2 であれば、その時の sytactic[].result に格納されている値に従って、以下のようなコードが生成される。

Ⅰ）sytactic[].result=='4'または'5'の時、

```
コード  V2      （コードは sytactic[].result の値に従う）
```

Ⅱ）sytactic[].result=='6'または'7'の時、

```
STORE   T
LOAD    V2
コード  T       （コードは sytactic[].result の値に従う）
```

Ⅲ）sytactic[].result=='3'の時、

STORE　V2

また、ここでも syntactic_analysis_1()関数と同様に、行数によるアドレスの重複を防ぐために、以下のコードにより上のコード生成におけるアドレスＴに対応するアドレスを、次の行に進むに従って十分に進めている。

```
if(w!=0){
        for(gyou2=0;gyou2<=w*MAXLENG;gyou2++){
                str++;
        }
}
```

（５）、メインルーチン
最後にメイン関数の説明を行う。
引数にとる対象言語のテキストファイルの存在の有無を調べ、エラーを出力する。
また、字句解析、構文解析、コード生成の関数を呼び出している。
そして、対象言語のテキストファイルの EOF を読み込んだら、解析を完了とし、

It succeeded in making 'compiler_data.txt'.

を、プロンプト上に出力し、'compiler_data.txt'をコンパイラと同じフォルダ内に出力する。
記号表、定数表、対象言語を字句で表示したもの、対象言語を構文解析したもの、アセンブリ言語で書かれたコードが、この'compiler_data.txt'に記述されている。

その例をここに記述しておく。

↓ここから

【1-gyoume】
【motono-siki】
i1 := num1 * i2 - i3

【gyakupooranndokihou-siki】
i1 num1 i2 * i3 - :=

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
abc	i1	00412906
e3	i2	00412A0E
def	i3	00412B16

teisu1	teisu2	address
256	num1	0041298C

【code-generation】

LOAD	0041298C
MUL	00412A0E
SUB	00412B16
STORE	00412906

↑ここまで

２、実行方法

コマンドプロンプトにより、compiler.c をコンパイルする。

コンパイルによって作成された実行ファイル（自分が行った環境では'compiler.exe'）を実行する。

その実行の際には、対象言語が入力されているテキストファイル（自分が行った環境では'siki_1.txt'）を引数にとる。

実行が終了すると、'compiler_data.txt'というテキストファイルが出力される。

'compiler_data.txt'の中に、プログラムの実行結果が記述されている。

※以下に、実際に自分の環境で実行したコマンドプロンプト画面を記述しておく。

↓ここから

C:¥Documents and Settings¥i865gpen4¥デスクトップ¥提出レポ（３回後期）¥コンパイラ動作試験>bcc32 compiler.c

```
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
compiler.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

C:¥Documents and Settings¥i865gpen4¥デスクトップ¥提出レポ（３回後期）¥コンパイ
ラ動作試験>dir
 C:¥Documents and Settings¥i865gpen4¥デスクトップ¥提出レポ（３回後期）¥コンパ
イラ動作試験 のディレクトリ

2006/01/29  01:26    <DIR>          .
2006/01/29  01:26    <DIR>          ..
2006/01/29  01:19            13,890 compiler.c
2006/01/29  01:26            62,976 compiler.exe
2006/01/29  01:26             9,329 compiler.obj
2006/01/29  01:26           393,216 compiler.tds
2006/01/28  23:32                39 siki_1.txt
2006/01/28  23:41                27 siki_2.txt
2006/01/29  00:10               126 siki_3.txt
               7 個のファイル             479,603 バイト
               2 個のディレクトリ  29,980,884,992 バイトの空き領域

C:¥Documents and Settings¥i865gpen4¥デスクトップ¥提出レポ（３回後期）¥コンパイ
ラ動作試験>compiler siki_1.txt

It succeeded in making 'compiler_data.txt'.

C:¥Documents and Settings¥i865gpen4¥デスクトップ¥提出レポ（３回後期）¥コンパイ
ラ動作試験>dir
 C:¥Documents and Settings¥i865gpen4¥デスクトップ¥提出レポ（３回後期）¥コンパ
イラ動作試験 のディレクトリ

2006/01/29  01:27    <DIR>          .
2006/01/29  01:27    <DIR>          ..
2006/01/29  01:19            13,890 compiler.c
2006/01/29  01:26            62,976 compiler.exe
2006/01/29  01:26             9,329 compiler.obj
2006/01/29  01:26           393,216 compiler.tds
2006/01/29  01:27               986 compiler_data.txt
2006/01/28  23:32                39 siki_1.txt
2006/01/28  23:41                27 siki_2.txt
```

```
2006/01/29  00:10               126 siki_3.txt
               8 個のファイル                 480,589 バイト
               2 個のディレクトリ  29,980,880,896 バイトの空き領域
```

↑ここまで

３、動作例とその説明

（１)、siki_1.txt

```
x := 3;
y := 5;
z := x * y - (y - x);
```

この対象言語は先生にもらった資料に記述されているもので､まずはこれをコンパイルにより解析してみる。
今回作ったコンパイラは、同じつづりのものを同じ字句として解析することができないため、同じであってもそれぞれ前から読んでいき、順次別の字句とアドレスを設定していっている。
以下の例では、

```
z := x * y - (y - x);
```

のｘとｙは、それぞれ式の中に２回ずつ使われているが、今回のプログラムでは前から順に以下のようにアドレスを設定しているのがわかる。

kigou1	kigou2	address
z	i1.3	004128DC
x	i2.3	00412960
y	i3.3	004129E4
y	i4.3	00412B70
x	i5.3	00412BF4

このように、したことでメモリをムダに使っている感が否めないが、計算の結果は同様のものが得られるためによしとした。

しかし、字句解析としてはこれは不十分な場所だといえる。
時間があれば、これも同じつづりのものは同じ字句・同じアドレスとして設定するようにしたい。

また、資料の 4.2.3 の課題 3-1 の問題も１行目と２行目のコード生成を見る限りでは、クリアしていることが分かる。

とりあえず、出力されたテキストファイルのコンパイラのデータの説明を個別にしておく。

（ｉ）【1-gyoume】
読み込んだ対象言語のテキストファイルの１行目を示している。
つまり、この動作結果の場合は１行目の解析は

'x := 3;'

の解析を示している。

（ii）【motono-siki】
読み込んだ対象言語を字句解析により解析し、その解析した字句で元の対象言語を書き直したものになっている。
１行目の解析を例にあげると、

x → i1.1
3 → num1.1

と字句解析し、対象言語を解析した字句と置き換えて

i1.1 := num1.1

と表現している。

（iii）【gyakupooranndokihou-siki】
字句解析した対象言語を、構文解析により逆ポーランド記法で表したものに書き換えている。
１行目の解析を例にあげると、元の対象言語が以下なので、

i1.1 := num1.1

これを逆ポーランド記法で書き換える事で、以下のように表現される。

i1.1 num1.1 :=

（ⅳ）【kigouhyou/teisuuhyou】
ここで、字句解析によって解析された識別子と定数の表を表示している。
１行目の例では、上記で説明した通り、

x → i1.1
3 → num1.1

と字句が表現し直されているので、以下のような記号表と定数表が表示されている。

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
x	i1.1	004128DA

teisu1	teisu2	address
3	num1.1	00412960

ここで、

kigou1 → 元の対象言語におけるつづり
kigou2 → 字句解析によって表現し直された字句
adress → 字句解析によって設定された字句が格納されているメモリ上のアドレス（パソコン環境によって当然変わる）

となっている。

（ⅴ）【code-generation】
コード生成の結果が表示されている。
コードはアセンブリ言語で表現されており、アドレスは記号表・定数表で設定されているメモリ上のアドレスに対応している。
１行目のコード生成を例にあげると、コードが、

LOAD	00412960
STORE	004128DA

となっていて、アドレスと字句の対応が、

kigou1	kigou2	address
x	i1.1	004128DA

teisu1	teisu2	address
3	num1.1	00412960

となっているので、
まず、'LOAD 00412960'により、ACCにアドレス'00412960'上のデータ、記号表より'3'が置かれる。
次に、'STORE 004128DA'により、ACC 上のデータつまりは、'LOAD 00412960'によって読み込まれた'3'が、アドレス'004128DA'のメモリに読み込まれる。
このときアドレス'004128DA'は、'x'が置かれているために、'x := 3;'という意味になり、マシンが'x := 3;'という対象言語をアセンブリ言語により理解したことになる。

ここで、個別に各コードの説明をしておく。

命令コード		意味
LOAD	M	(M) → ACC
STORE	M	(ACC) → M
ADD	M	(ACC) + (M) → ACC
SUB	M	(ACC) - (M) → ACC
MUL	M	(ACC) * (M) → ACC
DIV	M	(ACC) / (M) → ACC

※()はそのアドレス上に置かれているデータの事を示している。

【siki_1.txt】の動作結果をみると、上記に説明した内容を表現できているようだ。

↓【siki_1.txt】の動作結果ここから

【1-gyoume】
【motono-siki】
i1.1 := num1

【gyakupooranndokihou-siki】
i1.1 num1.1 :=

【kigouhyou/teisuuhyou】

kigou1	kigou2	address

```
x               i1.1            004128DA

teisu1          teisu2          address
3               num1.1          00412960

【code-generation】
LOAD            00412960
STORE           004128DA



【2-gyoume】
【motono-siki】
i1.2 := num1.2

【gyakupooranndokihou-siki】
i1.2 num1.2 :=

【kigouhyou/teisuuhyou】
kigou1          kigou2          address
y               i1.2            004128DB

teisu1          teisu2          address
5               num1.2          00412963

【code-generation】
LOAD            00412963
STORE           004128DB



【3-gyoume】
【motono-siki】
i1.3 := i2.3 * i3.3 - (i4.3 - i5.3)

【gyakupooranndokihou-siki】
i1.3 i2.3 i3.3 * i4.3 i5.3 - - :=

【kigouhyou/teisuuhyou】
kigou1          kigou2          address
```

```
z               i1.3            004128DC
x               i2.3            00412960
y               i3.3            004129E4
y               i4.3            00412B70
x               i5.3            00412BF4

teisu1          teisu2          address

【code-generation】
LOAD            00412960
MUL             004129E4
STORE           00412D71
LOAD            00412B70
SUB             00412BF4
STORE           00412D72
LOAD            00412D71
SUB             00412D72
STORE           004128DC
```

↑【siki_1.txt】の動作結果ここまで。

（２）、siki_2.txt

```
abc := e3 * 256 - abc / e3;
```

これは、指導書に書かれている例の対象言語である。
相変わらず同じつづりのものを同じ識別子・定数として表現はできていないが、それ以外の問題はないと思われる。

一時的にデータを置いて置くアドレス（以下の例でいうと'00412BEE'や'00412BED'など）も、記号表と定数表で設定されている字句のアドレスと重複されることなく設定されているのがわかる。

↓【siki_2.txt】の動作結果ここから

```
【1-gyoume】
【motono-siki】
```

```
i1.1 := i2.1 * num1.1 - i3.1 / i4.1
```

【gyakupooranndokihou-siki】

```
i1.1 i2.1 num1.1 * i3.1 i4.1 / - :=
```

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
abc	i1.1	004128DA
e3	i2.1	0041295E
abc	i3.1	00412AEA
e3	i4.1	00412B6E

teisu1	teisu2	address
256	num1.1	004129E4

【code-generation】

LOAD	0041295E
MUL	004129E4
STORE	00412BED
LOAD	00412AEA
DIV	00412B6E
STORE	00412BEE
LOAD	00412BED
SUB	00412BEE
STORE	004128DA

↑【siki_2.txt】の動作結果ここまで。

（３）、siki_3.txt

```
abc := def * 256 - e8;
abc := def * 256 - (e8 + 512);
abc := def * 256 + (e8 + 512);
abc := def * 256 / (e8 + 512);
abc := def * 256 * (e8 + 512);
```

【siki_3.txt】では、スタック上の ACC の位置によるコード生成の場合分けを全てふまえたような対象言語を設定して、全ての場合において、正確にコード生成ができているかを調べてみた。

指導書における、表４の命令コード生成表を参考にした。

１行目は命令コード生成表の１、２、５の命令を使って生成している。生成されたコードをみると正確に生成されている。

２行目は命令コード生成表の１、４、５の命令を使って生成している。生成されたコードをみると正確に生成されている。

３行目は命令コード生成表の１、３、５の命令を使って生成している。生成されたコードをみると正確に生成されている。

４行目は命令コード生成表の１、４、５の命令を使って生成している。生成されたコードをみると正確に生成されている。

５行目は命令コード生成表の１、３、５の命令を使って生成している。生成されたコードをみると正確に生成されている。

↓【siki_3.txt】の動作結果ここから

【1-gyoume】
【motono-siki】
i1.1 := i2.1 * num1.1 - i3.1

【gyakupooranndokihou-siki】
i1.1 i2.1 num1.1 * i3.1 - :=

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
abc	i1.1	00412906
def	i2.1	0041298A
e8	i3.1	00412B16

teisu1	teisu2	address

256 num1.1 00412A10

【code-generation】

LOAD 0041298A
MUL 00412A10
SUB 00412B16
STORE 00412906

【2-gyoume】

【motono-siki】

i1.2 := i2.2 * num1.2 - (i3.2 + num2.2)

【gyakupooranndokihou-siki】

i1.2 i2.2 num1.2 * i3.2 num2.2 + - :=

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
abc	i1.2	00412907
def	i2.2	0041298B
e8	i3.2	00412B9B

teisu1	teisu2	address
256	num1.2	00412A13
512	num2.2	00412C23

【code-generation】

LOAD 0041298B
MUL 00412A13
STORE 00412D1D
LOAD 00412B9B
ADD 00412C23
STORE 00412D1E
LOAD 00412D1D
SUB 00412D1E
STORE 00412907

【3-gyoume】

【motono-siki】

i1.3 := i2.3 * num1.3 + (i3.3 + num2.3)

【gyakupooranndokihou-siki】

i1.3 i2.3 num1.3 * i3.3 num2.3 + + :=

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
abc	i1.3	00412908
def	i2.3	0041298C
e8	i3.3	00412B9C

teisu1	teisu2	address
256	num1.3	00412A16
512	num2.3	00412C26

【code-generation】

LOAD	0041298C
MUL	00412A16
STORE	00412D9D
LOAD	00412B9C
ADD	00412C26
ADD	00412D9D
STORE	00412908

【4-gyoume】

【motono-siki】

i1.4 := i2.4 * num1.4 / (i3.4 + num2.4)

【gyakupooranndokihou-siki】

i1.4 i2.4 num1.4 * i3.4 num2.4 + / :=

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
abc	i1.4	00412909
def	i2.4	0041298D
e8	i3.4	00412B9D

teisu1	teisu2	address
256	num1.4	00412A19
512	num2.4	00412C29

【code-generation】

LOAD	0041298D
MUL	00412A19
STORE	00412E1D
LOAD	00412B9D
ADD	00412C29
STORE	00412E1E
LOAD	00412E1D
DIV	00412E1E
STORE	00412909

【5-gyoume】

【motono-siki】

i1.5 := i2.5 * num1.5 * (i3.5 + num2.5)

【gyakupooranndokihou-siki】

i1.5 i2.5 num1.5 * i3.5 num2.5 + * :=

【kigouhyou/teisuuhyou】

kigou1	kigou2	address
abc	i1.5	0041290A
def	i2.5	0041298E
e8	i3.5	00412B9E

teisu1	teisu2	address
256	num1.5	00412A1C
512	num2.5	00412C2C

【code-generation】

LOAD	0041298E
MUL	00412A1C
STORE	00412E9D
LOAD	00412B9E

```
ADD             00412C2C
MUL             00412E9D
STORE           0041290A
```

↑【siki_3.txt】の動作結果ここまで。

（４）、エラー表示・不完全な部分

・対象言語に式の終わりを表す記号';'がないと、無限ループになってしまう。
・(() のような対象言語の場合でも、(() → () と判断し解析してしまう。
・':'の次に'='がこなかったとき字句解析エラーを表示する。
・逆ポーランド記法において、最初に識別子か定数がこなかった場合、コード生成エラーを表示する。

４、コンパイラ検討

（１）、字句解析による誤り処理

字句解析における誤り処理としては、読み込む文字が未知の場合におこることがある。
処理できる言語において、定義していない字句を読み込んでしまった場合、この字句解析は適したトークンを返せない。

（２）、コンパイラに用いられるほかの構文解析法について。

作成したコンパイラのプログラムでは､LL(1)文法において､下向き構文解析を再帰的に用いるものであった。その他には次のような構文解析がある。

（ｉ）、シフト還元構文解析

入力文字を先頭からスタックに読み込んでいき、構文解析表を参考にして、順次文法規則で還元していく、上向き構文解析。特徴は最右導出の逆順で解析する手法である。最右導出とは、常に最も右の非終端記号を生成規則によって書き換えることが出来る。

（ⅱ）、LR 構文解析

シフト還元構文解析のうちで最も一般的で、効率がよいものである。解析できる文法も LL(1)文法よりも多く、決定性構文解析のうちでは最も広い範囲を解析することが可能。左から右に解析して分かる最も早い段

階で構文誤りを検出できる。LR 構文解析では、ある時点まで読んで分かる複数の可能性を状態として覚えておく。

（iii）、SLR(1)構文解析
ＬＲ構文解析の中で最も単純で、還元のために先読み記号の決め方は follow を用いる。構文解析表は、LR(0)集成を用いる。

（iv）、正準 LR(1)構文解析
SLR(1)構文解析に対し、正準 LR(1)集成から構文解析表を作る。SLR(1)より状態を細分化することにより衝突を回避する。それによって、状態数は多くなる。一つ先の先読み記号を持つために、SLR(1)より多くの情報を保持する事が出来る。

（ｖ）、LALR(1)構文解析
これは、正準 LR(1)の状態数をへらしたものである。解析できる文法は正準 LR(1)と等しい。

現在、実用化されているコンパイラのほとんどがこれらの LR 構文解析であるようだ。

（３）、意味解析と最適化
（ｉ）、意味解析
意味解析とは、プログラムソースに出てきた１つの意味が、そのプログラムを通して、一貫しているかどうかの意味規則の対応を取ることである。
例えば、Ｃ言語において、int で宣言した変数は、そのプログラムでは終始 int としてしか使えない、こういった変数や、定数などの情報を管理するために記号表を作る。
また、これ以外に、変数のスコープの管理も意味解析に含まれる。ブロックが入れ子になったとき、深いブロックにある変数は、そのブロックより浅いブロックからは範囲外である。スコープ管理も記号表にブロックレベルの欄を作ることによって、制御する事が出来る。
コンパイラの扱える文法が複雑になってしまったり、プログラムソースが長くなるにつれて、意味解析での記号表は膨大な大きさになる。そのために、記号表の検索方法として、線形検索や２分法、ハッシュ法など、工夫されるのが一般的である。

（ii）最適化
最適化とは、プログラムの実行時間を短くする目的で、定数同士の掛け算や、同じ計算の省略、実行前からの害損の条件分岐など、コンパイル時に簡略することで、実行効率をあげることである。中間コードの形としては、三番地コードや四つ組などが有名である。コンパイラによっては、何度も最適化し、中間コードを生成するものもあるが、一般的に最適化はコンパイラの要であるために、そう方法については、非公開なこともある。

（４）、条件文、繰り返し文に対するコード生成
２重の if 文

```
if(e1){
        if(e2)s1 else s2
}
```

に対して、以下のようなコード生成が得られる。

```
---------------------------------------
e1 を計算し、結果が偽なら L1 へ移動する。
e2 を計算し、結果が偽なら L2 へ移動する。
S1 の実行
jmp L3
L2：s2 の実行
L3：
L1：
---------------------------------------
```

L1 と L3 はそれぞれ、外側の内側の if 文のコードの終わりを表すラベルである。これらは同じ場所を表すので、一方のラベルだけを使い、以下のようにするほうが、より簡潔のなる。

```
---------------------------------------
e1 を計算し、結果が偽なら L1 へ移動する。
e2 を計算し、結果が偽なら L2 へ移動する。
S1 の実行
jmp L1
L2：s2 の実行
L1：
---------------------------------------
```

（５）、字句解析器生成系と構文解析器生成系
（ｉ）、字句解析器生成系
字句解析器生成系とは、文法ファイルに世紀表現で、処理対象の字句の定義を書いておく事で、自動的に字句解析プログラムを生成してくれるツールのことを言う。
有名なのが、lex とよばれ、yacc（次に説明する構文解析器生成系のツール）で生成する構文解析プログラムが呼び出す字句解析プログラムを lex が自動生成する。文法ファイルの記述は、正規表現で字句を記述する。

（ⅱ）、構文解析器生成系

構文解析器生成系とは、字句解析器生成系と同様に、記号列だけを見て会席できるように言語を設計し、自動的に構文木をつくるツールのことである。最も有名なものとして、yacc というのがある。yacc は、文法ファイルという自分の扱いたい文法を記述したファイルを、yacc に入力することで、自動的にＣ言語のソースコードを生成してくれる。yacc では、上向き構文解析で奇異席するので、今回の実験の LL(1)文法ではエラーが発生するために使えない。文法ファイルは、３つの部分からなっており、ヘッダ、規則部、ユーザー定義部である。最も重要な部分は規則部で、ここにＢＮＦで対象言語を書くと、後は自動で解析してくれる。

５、実験に対する感想や独自の考察

今回は前期よりもさらに膨大なプログラムを作らなければならないため、よりプログラミングに対する深い理解をする必要があった。
最初はできるか不安だったが、やっているうちに楽しくなり、なんとかここまで作ることができた。

このプログラム作成のおかげで、Ｃ言語に対する深い関心を持てたと思う。